SCIENCE CHINA
Information Sciences

• **RESEARCH PAPER** •

May 2017, Vol. 60 052110:1–052110:17
doi: 10.1007/s11432-016-5521-0

Accurate and efficient exploit capture and classification

Yu DING[1], Tao WEI[2], Hui XUE[2], Yulong ZHANG[2], Chao ZHANG[2] & Xinhui HAN[1]*

[1]*Institute of Computer Science and Technology, Peking University, Beijing* 100080*, China;*
[2]*Eletrical Engineering and Computer Sciences , UC Berkeley, Berkeley, CA* 94720*, USA*

Received October 22, 2015; accepted November 24, 2015; published online September 13, 2016

Abstract Software exploits, especially zero-day exploits, are major security threats. Every day, security experts discover and collect numerous exploits from honeypots, malware forensics, and underground channels. However, no easy methods exist to classify these exploits into meaningful categories and to accelerate diagnosis as well as detailed analysis. To address this need, we present SeismoMeter, which recognizes both control-flow-hijacking, and data-only attacks by combining approximate control-flow integrity, fast dynamic taint analysis and API sandboxing schemes. Once it detects an exploit incident, SeismoMeter generates a succinct data representation, called an *exploit skeleton*, to characterize the captured exploit. SeismoMeter then classifies the captured exploits into different exploit families by performing distance computing on the extracted skeletons. To evaluate the efficiency of SeismoMeter, we conduct a field test using exploit samples from public exploit databases, such as Metasploit, as well as wild-captured exploits. Our experiments demonstrate that SeismoMeter is a practical system that successfully detects and correctly classifies all these exploit attacks.

Keywords software security, exploit classification, exploit attack capture, control flow integrity, JIT security

Citation Ding Y, Wei T, Xue H, et al. Accurate and efficient exploit capture and classification. Sci China Inf Sci, 2017, 60(5): 052110, doi: 10.1007/s11432-016-5521-0

1 Introduction

This study presents SeismoMeter, a system that captures and classifies incoming attacks based on their underlying exploits quickly and automatically in honeypot environments [1–4]. These environments include both server-side honeypots and honeyclients [5, 6].

A software exploit attacks victim software by taking advantage of software vulnerabilities to drive it toward an unexpected behavior. With the help of exploits, attackers can spread network worms, construct botnets, and initiate advanced persistent threat (APT) attacks [7].

In a concrete attack, attackers implement an exploit using malformed data and shellcode payload. A typical exploit comprises several steps, as illustrated in Figure 1. In Step 1, the attacker feeds the exploit consisting of malformed data and shellcode payload to the victim program. The malformed data drives the victim toward a target vulnerability through legitimate control flow paths, usually causing the

* Corresponding author (email: hanxinhui@pku.edu.cn)

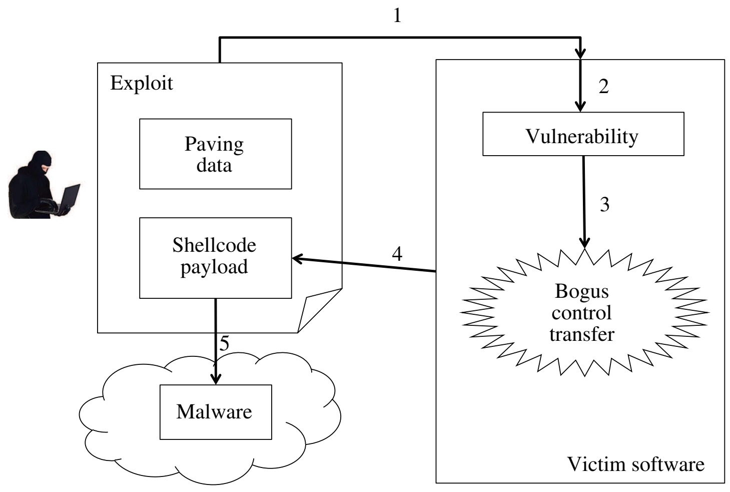


Figure 1 A typical workflow of exploits. Attackers rely on exploiting a target vulnerability to trigger bogus control flow transfer toward the shellcode payload, which may, in turn, further execute malware.

control data (e.g., a return address) to become tampered (Step 2). Thus, a bogus control transfer occurs (Step 3). The victim program then unexpectedly deviates from the legitimate control flow. It may crash and cause a denial-of-service (DOS) or start to run attacker's shellcode payload (Step 4). The shellcode can launch further attacks or simply download and execute some other powerful malware, as shown in Step 5.

Exploits play a major role in cyber attacks. Capturing and classifying exploits greatly improves understanding of the manner in which attacks launch. Captured exploits can reveal the details of vulnerabilities, as well as the skills employed in bypassing mitigation techniques, such as bypassing StackGuard [8], data execution pevention [9], and address space layout randomization [10]. Moreover, capturing and classifying exploits can help security researchers identify new vulnerabilities or new attack skills, and then improve protections such as generating fingerprints for intrusion detection/prevention systems, patching vulnerable systems, and developing more effective mitigation approaches. These capabilities are strongly demanded in CGC[1)], a recent state-of-art exploit attack race, and capture-the-flag contests.

Previous studies on exploit attack classifications fall into two categories: vulnerability-specific and network-level analysis. Vulnerability-specific analysis focuses on the routines of triggering vulnerabilities. Existing approaches such as [11–14] can be used to analyze exploits. However, in general, they are slow and infeasible for online exploit attack analysis (e.g., they may cause timeout of network interactions in honeypots). Network-level attack classification techniques characterize attacks at the network level. For example, WOMBAT [15] classifies attack events using statistical characteristics such as source IP, attack time and armies of zombies. Other approaches classify attacks based on characteristics of network packets [16–18]. However, these characteristics usually are not intrinsic properties of attacks. Thus attackers can bypass these approaches through evasion or obfuscation.

In this study, we present SeismoMeter, a novel exploit capturing and classification system, which is *efficient* and can be directly deployed in honeypots to monitor vulnerable programs and services online. It is also *accurate* in capturing and classifying real-world exploits. SeismoMeter characterizes exploits based on the exploit routine instead of vulnerability-specific characteristics. Therefore, SeismoMeter can recognize different exploits that target the same vulnerability. In addition, SeismoMeter can defeat network-level obfuscation and encryption because the classification process does not rely on network data.

To achieve accuracy when capturing, we identify both control-flow-hijacking and data-only attacks [19] (also known as non-control-data attacks [20]). While control-flow integrity (CFI [21]) provides a powerful method to detect control-flow hijacking attacks, it remains a challenge to model CFI in dynamically generated code and to detect automatically any violation of integrity. We propose *approximate Control Flow Integrity* (*aCFI*) to detect CFI-violation exploits. *aCFI* can be used to detect CFI violation not

1) Darpa Cyber Grand Challenge, https://cgc.darpa.mil/.

only in binary executables, but also in dynamically generated codes. We also develop fast dynamic taint analysis [22] and API sandboxing schemes to detect data-only attacks that do not violate CFI.

To achieve accurate classification, SeismoMeter must be sufficiently sensitive to differentiate exploits that target different vulnerabilities. It must also be able to differentiate exploits that targeting the same vulnerability but in different ways. However, SeismoMeter should also be sufficiently robust to tolerate various noises. This is necessary so the same exploits that attack different execution environments or inject different shellcode can be classified into the same group. To balance sensitivity and robustness, SeismoMeter generates an exploit skeleton for each exploit. The exploit skeleton is a concise representation that characterizes the manner in which an exploit hijacks a victim program. Exploits are then classified based on the distance between their exploit skeletons.

We implement SeismoMeter using 21 K lines of C code which run on Windows. We evaluate SeismoMeter's effectiveness using samples from both public exploit database Metasploit[2)] and real-world field tests. Experimental results show that SeismoMeter can classify exploits effectively and efficiently. During a seven-day field test, SeismoMeter captured 394 exploits and classified them into six categories. Of the 394 exploits, 255 targeted one of the most attacked vulnerability (MS08-067) [23] and these exploits comprise three families. We also monitored 5000 websites and captured 103 malicious drive-by downloading pages. SeismoMeter successfully classified them into 27 exploit families.

SeismoMeter contributes as follows:

• To the best of our knowledge, our study is the first to classify exploits into representative categories based on target victim program's execution trace.

• We propose a concise data representation called "exploit skeleton", to characterize exploits. Exploit skeleton captures the fundamental characteritics of the manner in which an exploit hijacks a program. We also devise a fast and accurate algorithm to classify exploits based on their exploit skeletons.

• We propose a scheme, *aCFI*, to detect CFI violation in both binary executables and dynamic generated codes. A novel approach that combines *aCFI*, dynamic taint analysis and API sandboxing schemes is implemented to detect both control-flow-hijacking and data-only attacks.

• We build a prototype of SeismoMeter, and evaluate it using both known and wild-captured exploits. Our experiments demonstrate that SeismoMeter can recognize and classify these exploit attacks (including just-in-time (JIT)-based attacks) efficiently and effectively. The field test shows that SeismoMeter is a practicable system.

The remainder of this paper is organized as follows. Section 2 provides an example that motivated our study. Section 3 details exploit skeleton which is the key of our study. Section 4 gives an overview of SeismoMeter and Section 5 describes the method SeismoMeter employs to capture and detect an exploit attack. Section 6 explains the manner in which SeismoMeter generates exploit skeletons and classifies them, and Section 7 describes our experiments and results. Section 8 reviews related works. We conclude our study in Section 9.

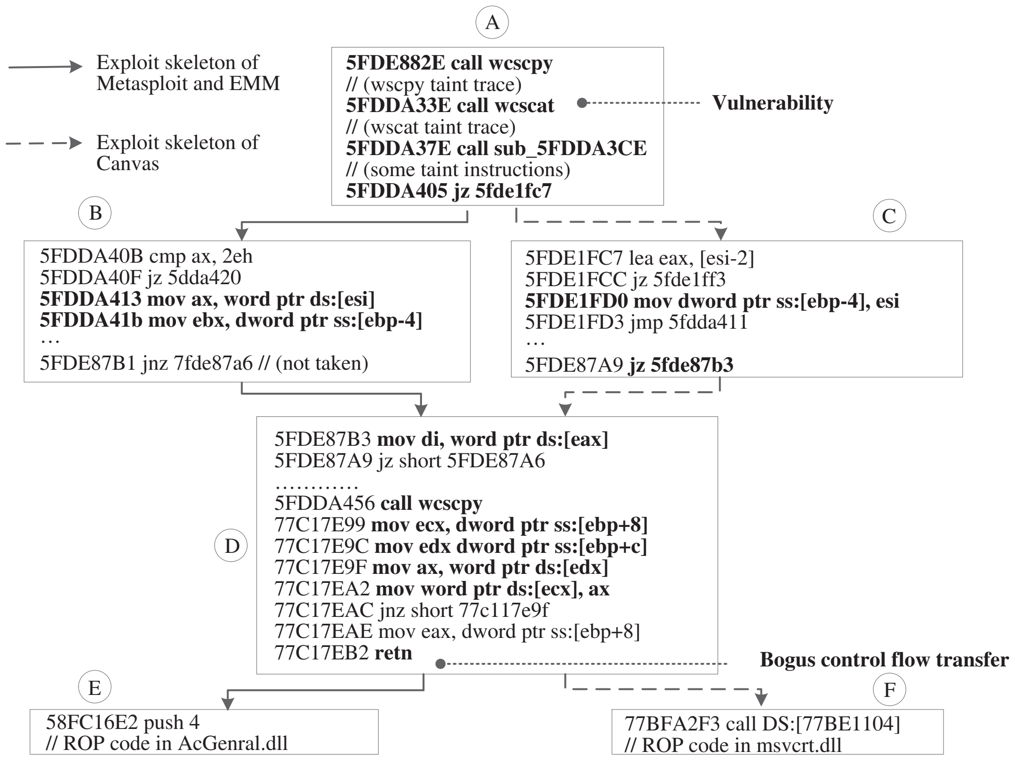
2 Motivating example

Given the same target vulnerability, different exploits can drive the victim program through different paths to achieve a bogus control transfer. We can differentiate these exploits by the paths they take.

One good case study is the massive attacks to MS08-067 , which comprised 61.2% of all software attacks in 2011. This vulnerability resides in the path handling function `NetPathCanonicalize` in `netapi32.dll`. An attack input with two "`..\`" as malformed data can trigger a stack overflow vulnerability through which the attacker can achieve a bogus control transfer toward shellcode payload.

We collected the known four attacks against this vulnerability from the exploit-db[3)]. These attacks were then conducted against the victim in a monitored environment. Figure 2 shows part of the execution traces.

2) Rapid 7, Metasploit, http://metasploit.com.
3) http://www.exploit-db.com.


Exploit skeleton of Metasploit and EMM
Exploit skeleton of Canvas
A
B
C
D
E
F
Vulnerability
Bogus control flow transfer

```
5FDE882E call wcscpy
// (wscpy taint trace)
5FDDA33E call wcscat
// (wscat taint trace)
5FDDA37E call sub_5FDDA3CE
// (some taint instructions)
5FDDA405 jz 5fde1fc7
```

```
5FDDA40B cmp ax, 2eh
5FDDA40F jz 5dda420
5FDDA413 mov ax, word ptr ds:[esi]
5FDDA41b mov ebx, dword ptr ss:[ebp-4]
…
5FDE87B1 jnz 7fde87a6 // (not taken)
```

```
5FDE1FC7 lea eax, [esi-2]
5FDE1FCC jz 5fde1ff3
5FDE1FD0 mov dword ptr ss:[ebp-4], esi
5FDE1FD3 jmp 5fdda411
…
5FDE87A9 jz 5fde87b3
```

```
5FDE87B3 mov di, word ptr ds:[eax]
5FDE87A9 jz short 5FDE87A6
…………
5FDDA456 call wcscpy
77C17E99 mov ecx, dword ptr ss:[ebp+8]
77C17E9C mov edx dword ptr ss:[ebp+c]
77C17E9F mov ax, word ptr ds:[edx]
77C17EA2 mov word ptr ds:[ecx], ax
77C17EAC jnz short 77c117e9f
77C17EAE mov eax, dword ptr ss:[ebp+8]
77C17EB2 retn
```

```
58FC16E2 push 4
// ROP code in AcGenral.dll
```

```
77BFA2F3 call DS:[77BE1104]
// ROP code in msvcrt.dll
```

Figure 2 Program traces for three attacks targeting the MS08-067 vulnerability. Arrows between boxes indicate control flow transfers that affect exploit attack routines. The bold font indicates a key instruction that is a type of exploit skeleton. Metasploit and EMM attack samples share the same exploit skeleton, which differs from that of the Canvas attack. For the same vulnerability, the three attacks can be classified into two categories based on their respective exploit skeletons.

From the execution traces, we found one major insight: the victim software executed identical key instructions (i.e., bold font instructions contained in blocks connected with solid lines) that contributed to the bogus transfer in both the Metasploit and EMM[4) samples. In contrast, in the Canvas[5) attack experiment, the victim software executed different key instructions (i.e., bold font instructions contained in blocks connected with dotted lines).

Each set of key instructions represents the manner in which an exploit works, including how the vulnerability is triggered, how the control data or other sensitive data are tampered, and the path that an exploit takes to achieve a bogus transfer. Therefore, such a set of key instructions can be used to distinguish exploits.

Based on this observation, we conclude that the Metasploit and EMM samples are indeed two attacks that implement the same exploit, even though they are written in different languages. However, the Canvas sample implements another exploit. This conclusion is confirmed by our manual analysis.

Given that so many attacks target MS08-067 and so few share available exploits, we can ease the jobs of security experts by building a SeismoMeter system that can map incoming attacks into families quickly and automatically. Security experts only need pay attention to those exploits that fall into new families and then analyze the exploits. In addition, we propose means to defend against new attacks.

3 Exploit skeleton

An exploit skeleton is a subset of a program execution trace that includes only the instructions that contribue to a bogus control transfer. We do not use the entire program execution for two reasons. First, the entire instruction trace of the program execution is sensitive to I/O events. A subtle I/O difference

4) EMM MS08-067 Universal Exploit, http://exploit-db.com/sploits/2008-MS08-067.rar.
5) Immunity Canvas Exploit, http://immunityinc.com.

may change program traces dramatically [24, 25].

Second, a program trace contains many instructions irrelevant to the attack. An instruction may relate to an attack in two ways: (1) It may be an instruction that navigates the control flow toward bogus transfer. Such instructions include branch instructions such as `br` and `jump` and function call instructions such as `call` and `ret`. We log `call` instructions when they recursively contain taint instructions and pair them with respective `ret` instructions. For branch instructions, we log all `jcc` branches and `loopcc` instructions. (2) It may be an instruction that propagates data tainted [22] by the attack input. Eventually, SeismoMeter collects all such instructions on the path of execution and outputs them as an exploit skeleton. Irrelevant instructions have no causal relationship with the attack and should be ignored in characterizing an attack.

In Figure 2, the traces mentioned in Section 2 are depicted using sequences of assembly code blocks. Metasploit and EMM attacks share the same trace (A→B→C→D), but which differs from the Canvas attack trace (A→E→C→F). In all three traces, the bogus control flow transfer occurs at line `0x77C17EB2`. They subsequently execute different malicious return oriented programming (ROP) gadgets [26]. Exploit skeleton instructions are highlighted using bold font. For each instruction in the exploit skeleton, we record: (1) the module name to which it belongs, (2) its offset within the module and (3) the binary instruction itself.

We create an exploit skeleton using a backward slicing algorithm [27] on the recorded program trace, which are decorated with taint tags that indicate the manner in which the input exploit affects the bogus control transfer. We also record information about function invoking and loop execution based on these tainted instructions as the major characteristics of control flow.

4 Design of SeismoMeter

4.1 Design principles

We employ the following design principles for SeismoMeter.

P1: Work on binaries. SeismoMeter should detect attacks effectively. To accomplish this, the detection should be able to capture not only static characteristics of the target vulnerable code, but also dynamic characteristics that are only available at runtime. In addition, SeismoMeter should function with proprietary software, and, thus, should work on binaries directly.

P2: Sensitive, robust attack classification. The classification process must extract essential information about an exploit and use them to classify the attacks. SeismoMeter must be able to characterize the manner in which an exploit hijacks the vulnerable program, and discard other irrelevant noises in the vulnerable program execution. The irrelevant noises here refer to instructions that do not participate directly. The optimal exploit features do not contain any irrelevant noises and only consist of instructions that directly participate in exploits.

P3: Low overhead. To become a practical system, SeismoMeter must have low overhead to be deployed in attack detection systems such as honeypots. More specifically, the overhead for exploit detection and classification should be sufficiently low to avoid session timeout.

4.2 SeismoMeter architecture

Figure 3 gives an overview of the SeismoMeter architecture. SeismoMeter consists of three main components: *pre-analysis*, *detection engine*, and *classification engine*.

Before the vulnerable binary is run, *pre-analysis* extracts a list of legitimate transfer targets using a *binary preprocessor*. A JIT profile contains generators that may generate legitimate transfer targets using a *JIT profiler*. This information is given to the *detection engine* in Step 3. After the vulnerable software launches in Step 4, the *detection engine* monitors the software's execution using tracing and dumps the execution trace for building the exploit skeleton.

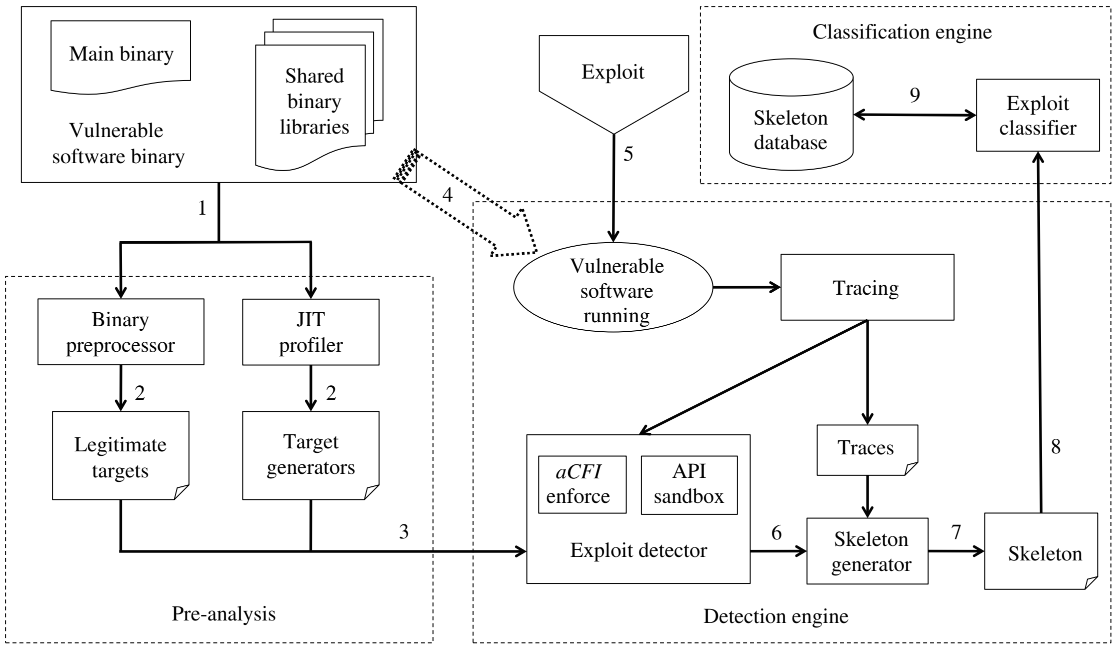


Figure 3 Overview of SeismoMeter architecture. The Preprocessor produces a whitelist by means of static binary analysis, which is used by Exploit Detector as detection criteria. When detecting an exploit, the Skeleton Generator generates an exploit skeleton to the Exploit Classifier, which then queries the Skeleton Database for alignment.

When an attack input arrives during Step 5, *exploit detector* captures the bogus transfer and notifies *skeleton generator* to calculate an exploit skeleton during Step 6. The *exploit classifier* in *classification engine* receives the exploit skeleton and performs classification based on known exploit skeletons in the database.

5 Detection methodology

SeismoMeter runs the victim program in a controlled environment and records its execution trace. In addition to tracing, SeismoMeter enforces the approximate Control Flow Integrity (*aCFI*) policy and sandboxes the execution of sensitive APIs. Once a runtime violation of *aCFI* or an unexpected call to sensitive API is detected, an alert is given regarding an exploit. Traditional whole system tracing solutions usually introduce more than 10 times the amount of normal overhead [22,28], which is excessive for online deployment. We port a PIN-based [29] fast dynamic taint analysis tool `libdft` [30] to Windows as "dftwin" and SeismoMeter benefits considerably from it.

5.1 Tracing execution efficiently with forward taint analysis

To analyze an exploit, we must be concerned about: (1) how the attack input affects data-flow and thus triggers the target vulnerability, (2) the path the program takes from entry to the vulnerable point and then to the bogus control transfer, and (3) the calling context when the attack succeeds.

SeismoMeter uses forward dynamic taint flow analysis to record all instructions that participate in propagating the input data and thus yield a much lower overhead. The taint attribute, which is originally attached to incoming attack input such as network packets, is propagated to the rest of the system along with the program execution. Instructions that have at least one tainted operand are logged to the trace.

In addition, for each tainted instruction, all branch instructions (e.g., *jnz* and *jmp*) from the current function entry to the current tainted instruction and then to the function exit are all dumped to the trace as well in order to reflect the control flow path that the exploit takes.

Moreover, for each function invocation, if at least one tainted instruction exists in the callee function's execution, we dump the corresponding *call* and *ret* instructions to the trace. Otherwise, this function call is not relevant to the taint propagation and dumping can be skipped. All remaining unmentioned instructions are not included in the trace.

Tracing stops when an exploit attack is detected (e.g., a violation of the *aCFI* policy is detected). Finally, the exploit skeleton is extracted from the trace.

5.2 *aCFI* enforcement

To detect control flow hijacking attacks, we monitor the execution of the vulnerable program during tracing and enforce a coarse-grained control flow integrity policy, that is, *aCFI*. Any violation of *aCFI* reflects a control flow hijacking attack at runtime. Our *aCFI* policy checks the control flow integrity for not only static code, but also JIT dynamic generated code.

Specifically, our *aCFI* enforcement consists of two forms of validation: (1) *indirect jump/call validation*, which examines the legitimacy of targets for indirect jumps and indirect function calls, and (2) *return address validation*, which strictly pairs the return address with its corresponding function call. The *exploit detector* component in SeismoMeter, as shown in Figure 3, is responsible for this enforcement. Other control flow transfer instructions, such as direct jumps, direct calls, and conditional jumps, are not validated at runtime, because their branch targets are fixed at compile time.

To conduct return address validation, when each return instruction is executed, we compare the return address on top of the real stack against that on top of the shadow stack. Three cases for this comparison are as follows:

• The return address on top of the real stack matches that on top of the shadow stack. It indicates that the control flow is not hijacked. The return address and the address of the corresponding call instruction can then safely popped out from the shadow stack.

• The return address on top of the real stack matches one return address in the shadow stack other than that on the top. In this case, the return address may be legal because some exception handling schemes work in this manner. In this situation, we pop out the shadow stack until the matched return address has also been popped out.

• The return address on top of the real stack mismatches all return addresses in the shadow stack. In this case, the control flow integrity has been violated and a successful exploit is detected and reported.

5.3 *aCFI* for JIT code

The control flow integrity of JIT dynamically generated code has not been well studied. Powerful attack techniques, such as JIT spraying [31] benefit from writable and executable memory attributes of dynamically generated code. These allow makes attackers to bypass the widely deployed protections such as address space layout randomization (ASLR) and data execution prevention (DEP) more easily.

To enforce the *aCFI* policy on dynamically generated code, we also must validate the transfer targets of all indirect call/jump and return instructions. Return instructions can be validated through a shadow stack scheme such as statically compiled code. However, for indirect call/jump instructions, we must overcome the challenge of gathering all legitimate transfer targets that are generated at runtime, and then validate them during execution.

Although some transfer targets of the indirect call/jump instructions are dynamically generated, the code snippets in the JIT engine that are responsible for generating such instructions are fixed and can be identified offline. We thus propose a novel two-step solution to solve this problem. First, we try to identify code snippets in the JIT engine that are responsible for generating transfer targets (e.g., function entries). These code snippets are referenced as *entry generators*. We then monitor the execution of these generators, and log all generated entries at runtime. In dynamic generated code, only these generated entries are legitimate transfer targets for indirect call/jump instructions.

Identifying entry generators: We have developed an offline program called *JIT profiler*, as shown in Figure 3, to address the aforementioned problem. The JIT profiler starts the target program, loads benign input data that may invoke the JIT engine (such as swf movies), generates a full instruction trace, and records the memory access for each instruction. We then scan the trace and identify all indirect

```
002319B0  mov dword ptr [ecx], 042444C7h
002319B6  mov byte ptr [ecx+8], 0E9h
002319BA  mov [ecx+9], eax
```

(a)

```
006589DC  mov  ebx, dword_B113C8
…
006589F2  or   bl, 50h
006589F5  mov  eax, edi
006589F7  pop  edi
006589F8  mov  [esi], bl
```

(b)

```
IonCode *
IonRuntime::generateEnterJIT(JSContext *cx,
EnterJitType type)
{
   MacroAssembler masm(cx);
   masm.push(ebp);
       …
}
inline nsresult push(void* aObject)
{
   return AppendElement(aObject) ? NS_OK :
NS_ERROR_OUT_OF_MEMORY;
}
```

(c)

Figure 4 Three types of "entry generators" in JIT engines. (a) G1; (b) G2; (c) G3.

jump/call instructions (in both static and dynamic code), whose targets fall into dynamically generated code areas. All of these transfer targets are legitimate dynamic code entries (e.g., function entries). Afterwards, we scan the trace again, and use the recorded memory access information to identify all instructions that write contents to the memory of these code entries. These instructions are called *entry generators*.

After profiling Microsoft Internet Explorer (IE), Adobe Flash and Firefox, we determine that only a few dozen generators exist in each JIT engine. These generators are of the following three types (shown in Figure 4).

• **Type G1:** `mov mem, imm`. In this type, an immediate value `imm` (i.e., a specific instruction) is written to the destination code entry `mem`. Figure 4(a) shows such an example in Flash. The generator here writes a fixed instruction (i.e., `042444C7h`) at address `ecx`.

• **Type G2:** `mov mem, fixed_reg`. In this type, the content of a fixed register is written to the destination memory `mem`. This type of generator can write code entries on demand. Figure 4(b) shows such a generator in Flash, in which the content of register `bl` is written to address of `esi`.

• **Type G3:** `mov mem, variable_reg`. A bundle of this type of generator is usually invoked by a wrapper function. Figure 4(c) shows an example in SpiderMonkey from Firefox. (The source code is listed here for easy reading.) The wrapper function `AppendElement` then invokes underlying generators according to its arguments. It is worth noting that this kind of generator is invoked frequently in JIT engines not only by those wrapper functions, but also by some other functions that also write to memory.

Recognize generated indirect targets: After identifying all entry generators, we monitor the execution of these generators to locate legal indirect transfer targets and enforce them at runtime.

When generators of type G1 and G2 are executed, the memory currently being written to is marked as a legitimate indirect transfer target. When generators of type G3 are executed, because they may be invoked by functions that do not write code entries, we must reconfirm that it is invoked by known wrapper functions and it is used to write legitimate code entries. Moreover, we monitor the execution of heap manipulating functions such as `free` to ensure indirect transfer target tables are updated.

Therefore, in the JIT profiler, we collect information including: (a) the value written to memory when these generators write code entries, and (b) the functions that call these generators to write code entries. At runtime, for a generator of type G3, we must confirm: (1) whether the written value equals any previously known value in the JIT profile, and (2) whether the caller (or caller's caller) matches any previously known caller. If and only if these two criteria are both satisfied do we mark the memory that is being written to as a legitimate indirect transfer target.

With information related to these dynamic transfer targets, the *exploit detector* in SeismoMeter can enforce the *aCFI* policy not only for statically compiled code but also for JIT generated code. If any indirect jump/call instruction attempts to jump to a JIT generated code area that does not belong to recognized legitimate indirect entries, an exploit attack alarm is raised.

5.4 Beyond *aCFI*: API sandboxing

Although *aCFI* is extremely effective against shellcode-injection and return-to-libc attacks, it is incapable of detecting call-to-libc attacks, including non-control-data attacks [20]. Such attacks usually do not tamper control data (e.g. return addresses) directly. Instead, they may taint some sensitive data in memory such as user IDs or commands to be executed in order to conduct attacks. Because non-control-data attacks do not hi-jack the control flow to execute injected codes, conventional CFI enforcement cannot detect such attacks. Furthermore, any exploit classification method based on conventional CFI enforcement cannot classify non-control-data exploits.

To detect these kinds of attacks, the *exploit detector* sandboxes the execution of sensitive APIs (e.g., `VirtualProtect` and `system`). First, it determines whether sensitive data (e.g., parameters) are tainted when these APIs are called. For example, if the target EIP is tainted, a typical control-flow hijacking is detected, or, if the argument of `system` function is controlled by attackers, an attack alarm is also raised. The *exploit detector* then performs the standard sandboxing checks to determine whether the monitored sensitive API must write to write a system file or system registry key, or create a process abnormally (i.e., that is not in the known legal whitelists).

6 Exploit skeleton generation and classification

As previously discussed, the exploit skeleton represents the manner in which the control flows are hijacked by the attacker step by step. In this section, we discuss the method SeismoMeter uses to generate such exploit skeletons and how to classify exploit skeletons.

6.1 Generation

After an exploit attack is detected, SeismoMeter stops tracing and begins extracting exploit skeletons from dumped traces.

The backward slicing algorithm [27] is used to generate an exploit skeleton. Starting from the bogus control transfer instruction, the algorithm tries to identify all key instructions and tainted target codes (e.g., injected shellcode). As shown in Figure 2, key instructions are those that influence the bogus transfer target (i.e., a return address or function pointer). Finally, the slicing algorithm obtains a subset of the instruction trace and builds the exploit skeleton.

JIT-generated code: For instructions in the JIT-generated dynamic code section, only the opcodes participate in the construction of the exploit skeleton. This is because instructions in JIT dynamic code are heavily position-dependent. Therefore, their addresses and operands may be quite different, even for instructions referring to the same exploit in different trials.

Loops: The loop structure is a major component of traces. In many exploit attack of memory corruption vulnerabilities, the attacker tampers the boundary of the iterator and gains access to additional memory. In program traces of such attacks, recognizing loop structures greatly helps to characterize exploits by pinpointing vulnerability loops. SeismoMeter recognizes loops using the algorithm from [32]. It iteratively identifies nested loops. Each time a loop is executed in the trace, only one instance of this loop is recorded, as well as the key instructions inside the loop body and the iteration count of this loop.

Skeleton size: The length of the skeleton affects the effectiveness and efficiency of our system. If the exploit skeleton is too short, it fails to characterize exploits. By contrast, a long exploit skeleton affects performance, especially for classification. In fact, 128 as the number of instructions, is long enough to differentiate all samples in our experiments. However, we prefer to extend the size and choose 512 as the skeleton size which does not generate additional performance overhead.

6.2 Classification

SeismoMeter classifies exploits into different families according to their skeleton similarities. Each skeleton represents a string of instructions. Therefore, classic distance-measuring algorithms for sequences can be applied for the purpose of exploit classification.

Specifically, we choose the classic Levenshtein distance [33] algorithm to compute similarities. However, other distance-measuring algorithms also should work. Classification is processed by comparing captured exploit skeletons to known skeletons. If the distance between the new and a previous skeleton is smaller than a threshold, the two exploits are treated as if members of the same exploit family. According to our experiments, different exploit families are vastly different. The threshold we choose is 15%. In other words, if the distance of any two skeletons is greater than 15% of the size of the skeleton, they are classified into different exploit families. Our experiment shows that threshold can distinguish different exploit attacks efficiently. More specifically, SeismoMeter compares the 512 instructions in two exploit skeletons and calculates their distances. When comparing instructions, three cases are used.

• **JIT-generated instructions:** Two JIT-generated instructions are equal to each other if and only if their opcodes are the same.

• **Instructions in loops:** Instructions in a loop are compared as a whole. Only the loops (rather than the actual instructions themselves) are compared. Two loops are equal to each other if and only if their loop bodies are equal and their iteration counts are the same.

• **Other instructions:** For other normal instructions in the skeletons, they are compared using their module names and their offsets in the modules.

Instructions of different cases are considered as different. If the final distance is smaller than the threshold (e.g., $15\% \times 512$), the two skeletons are classified into the same family.

7 Evaluation

We deployed SeismoMeter in both server-side and client honeypots to evaluate its accuracy. For the dataset, we used wild-captured exploit attacks, exploits from public exploit databases, and penetration tools. Unlike a network-level intrusion detection experimental setup, no known dataset exists that provides ground truth for exploit tests. We built testbeds and manually evaluated false positive and false negative rates. Finally, we recorded performance data for SeismoMeter and explained its practical use in attack detection systems, such as honeypots.

7.1 Honeypot field tests for vulnerable services

To test SeismoMeter with wild-captured exploits, we set up the field tests. Our test setup used a Cesar FTP Server 0.99g, an Xitami HTTP server and a tftpdwin 0.4.2 that we ran as vulnerable services on top of Windows XP SP3 (English) with Windows services and firewall disabled. SeismoMeter monitors `svchost.exe`, `server.exe` in Cesar FTP Server, `tftpd.exe` in tftpdwin and `xigui32.exe` in Xitami HTTP server. This experiment lasted for a single week and SeismoMeter captured 394 exploit skeletons in six families. The Xitami http server can be attacked easily with an over-sized http request and this http server was compromised 127 times. The Windows NetBIOS service is a popular victim in hacker communities. We captured 255 attacks against NetBIOS, and they belonged to three different families, containing 31, 3, 221 attacks, respectively. The Cesar FTP server attracted 10 attacks from the same family, whereas tftpdwin server attracted two attacks from the same family.

To determine whether SeismoMeter misclassified any of these attacks, we reconfirmed their exploit skeletons manually and confirmed the correctness of these classifications. In other words, SeismoMeter reported 0% false positive rates during the entire course of the experiment. We could not calculate the false negative rate of the experiment results, because no ground truth existed in our experimental setup.

7.2 Honeypot drive-by download tests

Drive-by download pages usually utilize JavaScript to launch advanced attacks. For example, their code is usually obfuscated and may be composed of self-modification code in the attack, both of which challenge any attack detection and analysis system. Therefore, testing SeismoMeter against drive-by download attacks is advisable.

Table 1 Drive-by download test results

Family #	Vulnerability	Count	Representative site
1	CVE-2010-0806	9	quanyisz.com
2	CVE-2010-0806	18	www.91linux.com
3	CVE-2010-0806	3	bisem.edu.pk
4	CVE-2010-0806	3	www.ruthenia.ru
5	CVE-2010-0249	3	q.fgjg1.cn
6	CVE-2010-0249	7	littlehanoi.cz
7	CVE-2010-0249	5	kjasoi.cn:6135
8	CVE-2010-0249	2	7ooobalrassam.jeeran.com
9	CVE-2010-0249	7	www.chinaui.com
10	CVE-2010-0249	7	yourchurchsolution.com
11	CVE-2010-0249	8	www.dvdstore-online.nl
12	CVE-2010-0249	4	bestofproductions.com
13	CVE-2010-0249	11	vip.45765.cn

For one month, we monitored landing pages from 5000 websites using a crawler infrastructure from our previous study [34]. Within these web pages, SeismoMeter detected 103 malicious drive-by download pages and identified the vulnerabilities they attacked. The classification results of SeismoMeter showed that these 103 attacks fall into 27 families.

Table 1 shows the corresponding experimental results. Because of space constraints, we list only the most representative results. In our experiment, CVE-2010-0249 and CVE-2010-0806 were the two most attacked vulnerabilities. We witnessed each of them being attacked by multiple different exploits. Our manual analysis confirmed that the classification results were accurate. The false positive rate remains zero and we were unable to calculate the false negative rate. Examining all 5000 websites manually and creating the ground truth for the experimental setup was rather difficult. In a subsequent analysis, we sampled 300 of 5000 pages and determined whether these pages contained malicious content. As expected, we discovered nothing malicious in these landing pages.

7.3 Effectiveness of detection

To test SeismoMeter's ability to detect exploits, we collected 16 exploit samples from public or business penetration tools, as well as one from Yang's report [35] and one data-only exploit attack sample from a wild captured drive-by download page, as listed in Table 2. We use these samples because: (a) they belong to different categories, (b) they are popular exploits and target mostly real-world vulnerabilities that have been attacked, and (c) we can obtain their source code and port them to our platform. In addition, to prove that SeismoMeter can handle unseen attacks, we created a vulnerable program and crafted an exploit for it. For this test, in addition to the exploit skeleton, we also recorded the original network data that triggered the attacks using port mirroring on the switch and dumping the flow on a separate server with `tcpdump`.

SeismoMeter successfully captured all these exploits without false negatives, as shown in Table 2. SeismoMeter also reveals the type of final bogus transfer for each exploit. The first column in the table lists the exploit's target vulnerability and the second column shows the sources of these exploits. The third column shows the victim programs' names. The second-to-last columns indicates whether SeismoMeter captured the exploit, and the last column shows the type of alerted bogus control transfer. In the data-only exploit detection experiment, SeismoMeter successfully recognized a data-only attack. Here, the generated exploit skeleton reveals the manner in which the exploit downloaded a given executable to the victim machine and executed it.

Table 2 Exploit test results

Vulnerability	Source	Victim program	Captured	Bogus control transfer
CVE-2008-2463	Metasploit	Internet Explorer 6	$\surd$	indirect call to bogus function
CVE-2006-2961	Metasploit	Cesar FTP Server 0.99g	$\surd$	return to bogus address
CVE-2006-4948	Metasploit	tftpdwin 0.4.2	$\surd$	return to bogus address
CVE-2007-5067	Metasploit	Xitami HTTP server v2.5b6	$\surd$	return to bogus address
CVE-2009-0075	Metasploit	Internet Explorer 7	$\surd$	indirect jump to bogus address
CVE-2009-0075	Canvas	Internet Explorer 7	$\surd$	indirect jump to bogus address (JIT)
CVE-2010-0806	Metasploit	Internet Explorer 6	$\surd$	indirect jump to bogus address
OSVDB-62134	Metasploit	EasyFTP Server 1.7.0.11	$\surd$	return to bogus address
CVE-2006-6184	Metasploit	Allied Telesyn TFTP Server 1.9	$\surd$	return to bogus address
CVE-2008-1610	Metasploit	Quick FTP Pro 2.1	$\surd$	return to bogus address
CVE-2008-4250	Metasploit	Windows Server Service	$\surd$	return to bogus address
CVE-2006-3439	Canvas	Windows Server Service	$\surd$	return to bogus address
CVE-2003-0352	Canvas	Windows Server Service	$\surd$	return to bogus address
CVE-2003-0818	Metasploit	Windows Server Service	$\surd$	return to bogus address
CVE-2013-0634	Canvas	Acrobat Flash	$\surd$	return to bogus address
CVE-2012-1529	Yang Yu [35]	Internet Explorer 9	$\surd$	GIFT attack
Composed	Composed	test_vuln	$\surd$	return to bogus address
CVE-2007-4105	Wild Captured	Internet Explorer 6	$\surd$	Data-only attack

Table 3 How skeleton length affects classification results

Length	Canvas vs. Metasploit		Poly. vs. Metasploit	
	Dist.	Norm Dist. (%)	Dist.	Norm Dist. (%)
1	0	0	0	0
4	0	0	0	0
16	2	12.50	0	0
64	44	68.75	12	18.75
128	103	80.46	75	58.60
512	485	94.70	430	83.98

Table 4 Classification results using different lengths

Length	Min.	Avge.	Norm Min. (%)	Norm Avge. (%)
1	0	0	0	0
4	0	0	0	0
16	2	3.44	12.50	21.50
64	9	21.29	14.06	33.27
128	21	73.20	16.40	57.18
256	146	195.25	57.03	76.26
512	402	435.60	78.51	85.08

7.4 Good length for an exploit skeleton

To classify exploits efficiently and effectively by comparing exploit skeletons, the length of exploit skeletons should be carefully determined. As an example, we evaluate the well-known MS08-067 exploits and identify an appropirate length of an exploit skeleton, as shown in Table 3.

We compare exploit skeletons generated by the Canvas and Metasploit samples. The first column in Table 3 lists different lengths of exploit skeletons, and the second and third columns show the edit distances and normalized edit distances, respectively, of Canvas and Metasploit exploits. The fourth and fifth columns list similar results comparing the Metasploit and polymorphous exploits[6]. In this case study, we determined that the last four instructions in exploit skeletons were the same in all runs. This means that we cannot use only the last few instructions to classify exploits.

We manually collected 31 exploit pair samples, and each sample contained two exploits that target the same vulnerability but in different families. Based on these samples, we calculated the pairwise edit distances. The results are shown in Table 4. The table shows that the last four instructions are always the same. However, the distances become distinct when the exploit skeleton length reaches 16 or longer. From the results, we learn that an exploit skeleton length of 128 and a 15% normalized edit distance are sufficient for the classification algorithm. Because the length of exploit skeletons also affects performance

6) Polymorphours MS08-067 Exploit http://www.exploit-db.com/exploits/7104.

Table 5 File transfer performance results

Software	Protocol	Mode	Vanilla run	SeismoMeter		nullpin	
			Time (s)	Time (s)	Overhead	Time (s)	Overhead
Cesar FTP	FTP	Server	6.84	9.51	1.39x	8.52	1.25x
tftpdwin	TFTP	Server	8.52	11.97	1.40x	9.63	1.13x
WinSCP	SFTP	Client	9.3	13.85	1.49x	12.41	1.33x
wget	HTTP	Client	7.38	9.1	1.23x	8.85	1.20x

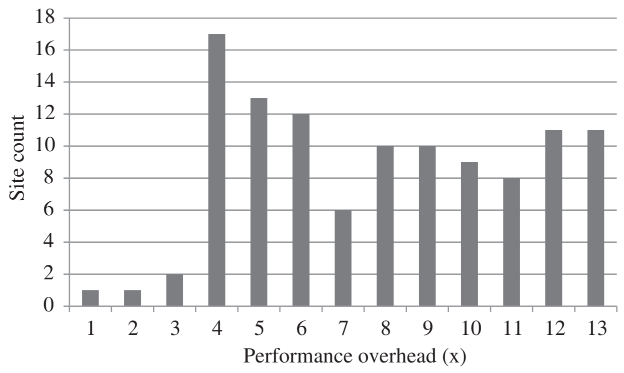


Figure 5 Web browsing performance result.

overhead, long lengths slow down SeismoMeter. However, SeismoMeter was proven to be sufficiently fast in real-world experiments. Therefore, we used 512 as the skeleton length and 15% as the threshold. We also determine that the exploit skeleton is extremely stable on the same exploit and is vastly different from exploit to exploit. This reveals that the seed of each family affects the class poorly. The chosen of seed exploit skeleton would not modify the results of the classification algorithm.

7.5 Performance evaluation

File transfer performance. We measure SeismoMeter's performance by means of file uploading and downloading. The single file we use is a randomly chosen 76-MB binary file. We measure the transfer time using three types of actions: (1) vanilla runs without SeismoMeter, (2) runs using SeismoMeter, and (3) monitoring system performance using PINs default mode `nullpin` [30]. Table 5 lists the results. The performance overhead of SeismoMeter is between 1.23x and 1.49x, which is fast for a dynamic taint-tracking system with a tracing ability. Existing tainting-analysis based detection approaches [22] such as Dytan [36] is 50x, and the instruction instrumentation mode of Ether [28] is more than 100x. SeismoMeter combines of dynamic taint analysis and *aCFI*. Thus, the performance overhead of SeismoMeter is incomparable to that of other CFI enforcements, because the performance overhead is mainly determined by taint analysis, instead of *aCFI* enforcement. The reason SeismoMeter is so fast is because it uses one of the fastest userland dynamic data-flow analysis tools [30] as a core component. Algorithms used in SeismoMeter also produced low performance overhead. Combining these algorithms enables SeismoMeter to be practical and deployable with good performance.

Browser benchmark. We script IE 9 running SeismoMeter to browse Alexa top 100 websites and presented the performance data in Figure 5. The X-axis indicates the performance overhead and the Y-axis indicates the number of sites that experience the corresponding amount of overhead. The websites that generate small performance overheads are mostly search engines, such as Google, and sites that show login pages. In addition, 11 websites were browsed that incur extremely high performance overhead. The maximum performance overhead is approximately 36x (qq.com). This site contains too much information and numerous ads and scripts. This performance reveals that SeismoMeter is practical for use in honeypot environments to capture attacks. Similar systems [36, 37] claim that performance overhead ranges from 30x to 1000x. Although browsing overhead occurs when analyzing a drive-by download page, we believe

SeismoMeter benefits security experts considerably as a result of its automatic analysis and scalable deployment. SeismoMeter neither claims to be a real-time threat protection system nor does it help security experts understand incoming attacks quickly. However, once deployed, SeismoMeter operates automatically, requiring no scalability for large deployments. Our study shows that a single SeismoMeter instance is capable of analyzing 30000–40000 page per day. A server with a moderate configuration can run 20 instances of SeismoMeter, which is acceptable in practical client-side honeypot systems.

8 Related work

8.1 CFI based exploit detection

CFI [38] is an important approach for improving software security and detecting attacks. Native Client [39] guarantees that indirect control flow will target an aligned address in a text segment, whereas SeismoMeter provides finer-grained and JIT-supported CFI protection. SeismoMeter collects legitimate jump targets by parsing PE files and is more precise than XFI [40] and BGI [41]. Existing CFI approaches often require that software source code to available. Hypersafe [42] enforces CFI for hypervisors. Hypersafe works as an LLVM [43] component and requires that the hypervisor recompiled. Control-Flow Locking (CFL) [44], limits the control flow by lock/unlock on branch/call/ret instructions. In addition, it must then recompile the program to protect CFI. By contrast, in terms of exploit detection and classfication, SeismoMeter works on exploit traces directly and does not require access to the exploit's source code.

8.2 Attack signature generation

Existing attack signature generation systems such as [3, 12, 45] comprise two approaches: (a) network-based, and (b) vulnerability based. Network based approaches such as [46] attempt to detect malicious network flows. However, they rely on static or emulation based shellcode detection algorithms that can be bypassed by some sophisticated attacks [16–18, 47]. By contrast, SeismoMeter reads instruction traces and focuses on those instructions that directly contribute to the bogus control transfer. This approach is immune to attacks using obfuscation and does not require a training process. Compared to Meta-Symploit [48], SeismoMeter does not require exploit's source code, which is hard to obtain in actuality. Brumley's work generates attack signatures using dynamic symbolic execution based analysis [49, 50]. The vulnerability-specific signature can describe precisely the triggering conditions of certain vulnerabilities. These kind of attack signatures can determine whether an input triggers a certain vulnerability. However, it cannot classify exploits, since malicious inputs generated by different exploits may share the same vulnerability-specific signature. Moreover, its performance overhead is comparatively high. SeismoMeter overcomes such problems effectively.

8.3 Exploit classification

The WOMBAT project [15] classifies exploits based on network data such as source IP and attack payloads. However, exploits may be encrypted or obfuscated at the network level [16–18]. PointerScope [14] provides automatic characterization of multi-stage exploits. It identifies major attack steps as pointer misuses by performing dynamic type inference on the execution of the attacked program, and then generates an attack graph to characterize the exploit. Although this approach is generally effective, PointerScope suffers from two drawbacks compared to SeismoMeter: (1) eliminating false positives is difficult because innocent type conflicts are prevalent during benign program execution, and (2) it is not efficient because it performs type checks and propagation during the execution of each single instruction.

Shield [46] classifies exploits by describing the state machine of a vulnerability. Shield recognizes exploit attacks over the transport layer. Manually written policy is required for the filtering process. A similar approach by Argos [1] provides exploit attack signature generation. However, unlike SeismoMeter, the signature generated by Argos only concerns the network package characteristics. Polygraph [51] characterizes exploits by recognizing the invariants of exploits and generating network based signatures.

ARBOR [52] also generates signatures from network packets and focuses only on buffer overflow vulnerabilities. Liang's follow-up study [53] uses a similar idea and generates signatures based on message length and contents. Compared to these other studies, ours is not limited to certain vulnerabilities, and is not limited to analyzing network-based signatures.

9 Conclusion

In this study, we presented SeismoMeter, an accurate and efficient exploit capturing and classifying system. SeismoMeter work on binaries directly by combining static binary analysis and dynamic binary instrumentation technique. SeismoMeter enforces the *approximate CFI* extracted from binary code modules, and provides JIT dynamic CFI validation using a novel approach. In addition, to detect accurately and robustly classify exploits, SeismoMeter computes and performs alignment checking on exploit skeletons. Our experiment results with both public available exploit samples and wild-captured attack incidents demonstrate that SeismoMeter is effective in detecting and classifying real world exploits. We believe that SeismoMeter is a practical system and can be deployed in real world honeypots.

Acknowledgements This work was supported by National Natural Science Foundation of China (Grant Nos. 61402125, 61572149).

Conflict of interest The authors declare that they have no conflict of interest.

References

1 Portokalidis G, Slowinska A, Bos H. Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. In: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems. New York: ACM, 2006. 15–27

2 Bailey M, Cooke E, Watson D, et al. A hybrid honeypot architecture for scalable network monitoring. University of Michigan Technical Report CSE-TR-499-04. 2006

3 Kreibich C, Crowcroft J. Honeycomb: creating intrusion detection signatures using honeypots. ACM SIGCOMM Comput Commun Rev, 2004, 34: 51–56

4 Spitzner L. Honeypots: concepts, approaches, and challenges. In: Proceedings of the 45th Annual Southeast Regional Conference. New York: ACM, 2007. 321–326

5 Diebold P, Hess A, Schäfer G. A honeypot architecture for detecting and analyzing unknown network attacks. In: Proceedings of Kommunikation in Verteilten Systemen (KiVS). Berlin: Springer, 2005. 245–255

6 Nazario J. PhoneyC: a virtual client honeypot. In: Proceedings of the 2nd USENIX Conference on Large-scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More. Berkeley: USENIX Association, 2009. 6

7 Cole E. Advanced Persistent Threat: Understanding the Danger and How to Protect Your Organization. Massachusetts: Syngress, 2012. 18–25

8 Cowan C, Pu C, Maier D, et al. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In: Proceedings of the 7th Conference on USENIX Security Symposium. Berkeley: USENIX Association, 1998. 346–335

9 Microsoft Corp. Data Execution Prevention. Microsoft Knowledge Base KB875352. 2013

10 PaX Team. PaX Address Space Layout Randomization (ASLR). Pax Team Report. 2003

11 Crandall J, Su Z D. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In: Proceedings of the 12th ACM Conference on Computer and Communications Security. New York: ACM, 2005. 235–248

12 Li Z, Sanghi M, Chen Y, et al. Network-based and attack-resillient lenght signature generator for zero-day polymorphic worms. In: Proceedings of the 15th IEEE International Conference on Network Protocols. Calfornia: IEEE Computer Society, 2007. 164–173

13 Joshi A, King S, Dunlap G, et al. Detecting Past and Present Intrusions Through Vulnerability-specific Predicates. In: Proceedings of the 20th ACM Symposium on Operating Systems Principles. New York: ACM, 2005. 91–104

14 Zhang M W, Prakash A, Li X L, et al. Identifying and analyzing pointer misuses for sophisticated memory-corruption exploit diagnosis. In: Proceedings of the 19th Annual Network and Distributed System Security Symposium. Virginia: Internet Society, 2012

15 Dacier M, Leita C, Thonnard O, et al. Cyber Situational Awareness. Berlin: Springer, 2010. 130–136

16 Fogla P, Sharif M, Perdisci R, et al. Polymorphic blending attacks. In: Proceedings of the 15th USENIX Security Symposium. Berkeley: USENIX Association, 2006. 241–256

17 Gundy M, Balzarotti D, Vigna G. Catch me if you can: evading network signatures with web-based polymorphic worms. In: Proceedings of the 1st USENIX Workshop on Offesive Technologies. Berkeley: USENIX Association, 2007. 7

18 Bania P. Evading network-level emulation. Computing Research Repository, 2007. abs/0906.1

19 Szekeres L, Payer M, Wei T, et al. Sok: eternal war in memory. In: Proceedings of the 2013 IEEE Symposium on Security and Privacy. Washington DC: IEEE Computer Society, 2013. 48–62

20 Chen S, Xu J, Sezer E, et al. Non-control-data attacks are realistic threats. In: Proceedings of the 14th Conference on USENIX Security Symposium. Berkeley: USENIX Association, 2005. 12–24

21 Abadi M, Budiu M, Erlingsson U, et al. Control-flow integrity. In: Proceedings of the 12th ACM Conference on Computer and Communications Security. New York: ACM, 2005. 340–353

22 Schwartz E J, Avgerinos T, Brumley D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: Proceedings of the 31st IEEE Symposium on Security and Privacy. Washington DC: IEEE Computer Society, 2010. 317–337

23 Symantec Corporation. Internet security threat report. Symantec Corporation Technical Report. 2012

24 Dunlap G, King S, Cinar S, et al. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. In: Proceedings of Symposium on Operating Systems Design and Implementation. New York: ACM, 2002. 211–224

25 Xu M, Malyguin V, Sheldon J, et al. Retrace: collecting execution trace with virtual machine deterministic replay. In: Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation. New York: ACM, 2007. 4–24

26 Shacham H. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In: Proceedings of the 14th ACM Conference on Computer and Communications Security. New York: ACM, 2007. 552–561

27 Agrawal H, Horgan J, Krauser E, et al. Incremental regression testing. In: Proceedings of the Conference on Software Maintenance. Washington DC: IEEE Computer Society, 1993. 348–357

28 Dinaburg A, Royal P, Sharif M, et al. Ether: malware analysis via hardware virtualization extensions. In: Proceedings of 15th ACM Conference on Computer and Communications Security. New York: ACM, 2008. 51–62

29 Luk C, Cohn R, Muth R, et al. Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. New York: ACM, 2005. 190–200

30 Kemerlis V, Portokalidis G, Jee K, et al. libdft: practical dynamic data flow tracking for commodity systems. In: Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments. New York: ACM, 2012. 121–132

31 Blazakis D. Interpreter exploitation. In: Proceedings of the 4th USENIX Conference on Offensive Technologies. Berkeley: USENIX Association, 2010. 1–9

32 Wei T, Mao J, Zou W, et al. A new algorithm for identifying loops in decompilation, In: Proceedings of the 14th International Conference on Static Analysis. Berlin/Heidelberg: Springer-Verlag, 2007. 170–183

33 Levenshtein V. Binary codes capable of correcting deletions, insertions and reversals. Sov Phys Dokl, 1966, 10: 707–710

34 Chen K Z J, Gu G F, Zhuge J W, et al. WebPatrol: automated collection and replay of web-based malware scenarios. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security. New York: ACM, 2011. 186–195

35 Yu Y. DEP/ASLR bypass without ROP/JIT. 13th Annual CanSecWest Conference Report. 2013

36 Clause J, Li W C, Orso A. Dytan: a generic dynamic taint analysis framework. In: Proceedings of the 2007 International Symposium on Software Testing and Analysis. New York: ACM, 2007. 196–206

37 Tucek J, Newsome J, Lu S, et al. Sweeper: a lightweight end-to-end system for defending against fast worms. In: Proceedings of ACM SIGOPS/EuroSys European Conference on Computer Systems. New York: ACM, 2007. 115–128

38 Abadi M, Budiu M, Erlingsson U, et al. Control-flow integrity principles, implementations, and applications. ACM Trans Inform Syst Secur, 2009, 13: 1–40

39 Yee B, Sehr D, Dardyk G. Native client: a sandbox for portable, untrusted x86 native code. In: Proceedings of the 2009 30th IEEE Symposium on Security and Privacy. Washington DC: IEEE Computer Society, 2009. 79–93

40 Erlingsson U, Valley S, Abadi M, et al. XFI: software guards for system address spaces. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation, Berkeley: USENIX Association, 2006. 75–88

41 Castro M, Costa M, Martin J, et al. Fast byte-granularity software fault isolation, In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, New York: ACM, 2009. 45–58

42 Wang Z, Jiang X X. HyperSafe: a lightweight approach to provide lifetime hypervisor control-flow integrity. In: Proceedings of the 2010 31st IEEE Symposium on Security and Privacy. Washington DC: IEEE Computer Society, 2010. 380–395

43 Lattner C, Adve V. LLVM: a compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization. Washington DC: IEEE Computer Society, 2004. 75–86

44 Bletsch T, Jiang X X, Freeh V. Mitigating code-reuse attacks with control-flow locking. In: Proceedings of the 27th Annual Computer Security Applications Conference. New York: ACM, 2011. 353–362

45 Wang L J, Li Z C, Chen Y, et al. Thwarting zero-day polymorphic worms with network-level length-based signature generation. Trans Netw, 2010, 18: 53–66

46 Wang H J, Guo C X, Simon D R, et al. Shield: vulnerability-driven network filters for preventing known vulnerability exploits. In: Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications. New York: ACM, 2004. 193–204

47 Mason J, Small S, Monrose F, et al. English shellcode. In: Proceedings of the 16th ACM Conference on Computer and Communications Security. New York: ACM, 2009. 524–533

48 Wang R W, Ning P, Xie T, et al. Metasymploit: day-one defense against script-based attacks with security-enhanced symbolic analysis. In: Proceedings of the 22nd USENIX Conference on Security. Berkeley: USENIX Association, 2013. 65–80

49 Newsome J, Brumley D, Song D. Vulnerability-specific execution filtering for exploit prevention on commodity software. In: Proceedings of the 13th Symposium on Network and Distributed System Security. Virginia: Internet Society, 2005

50 Newsome J, Brumley D, Song D. Towards automatic generation of vulnerability-based signatures. In: Proceedings of the 27th IEEE Symposium on Security and Privacy. Washington DC: IEEE Computer Society, 2006. 2–16

51 Newsome J. Polygraph: automatically generating signatures for polymorphic worms. In: Proceedings of the 2005 IEEE Symposium on Security and Privacy. Washington DC: IEEE Computer Society, 2005. 226–241

52 Liang Z K, Sekar R. Automatic generation of buffer overflow attack signatures: an approach based on program behavior models. In: Proceedings of the 21st Annual Computer Security Applications Conference. Washington DC: IEEE Computer Society, 2005. 215–224

53 Liang Z K, Sekar R. Fast and automated generation of attack signatures: a basis for building self-protecting servers. In: Proceedings of the 12th ACM Conference on Computer and Communications Security. New York: ACM, 2005. 213–222